MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Artificial Intelligence Project
Memo No. 152

January 1968.

PDP-6 LAP

Jon L White

CONTENTS

INTRODUCTION

LAP is a LISP FEXPR (or FSUBR when compiled) which is executed primarily for its side effect -- namely assembling a symbolic listing into core as a machine language subroutine. As such, it is about the most convient and rapid way for a LISP user to add machine language primitives to the LISP system, especially if the functions in question are in a developmental stage and are reasonably small (e.g. 1-500 instructions). Also, the LISP compiler currently gives its result as a file of LAP code, which may then be loaded into core by LAP.

Virtually any function definition, whether by DEFPROP, LABEL, or LAP, is an extension of LISP's primitives; and as in any actual programming language, the side-effects and global interactions are often of primary importance. Because of this, and because of the inherently broader range of machine instructions and data formats, a function quite easily described and written in PDP-6 machine language may accomplish what is only most painfully and artificially written in LISP. One must, then, consider the total amount of code in each language to accomplish a given task, the amount of commentary necessary to clarify the intent of the task given the program (in this sense, LISP code rates very high -- a major benefit of the confines of lisp is that a good program serves as its own comment, and usually needs no further elucidation), and other considerations of programming convience.

Experience has shown that many such subroutines may be assembled by a small system, i.e. one such as the current LAP, without conditional assembly, macro, or sophisticated literal generation features. These latter three features are the major differences in language between LAP and MIDAS; the major operational differences are (1) LAP is one-pass and MIDAS is two, (2) LAP uses the LISP READ function while MIDAS is more efficient, and (3) LAP assembles directly into the binary program space of the LISP system using it while MIDAS files its assembly on a peripheral device (which must then be loaded by STINK or the ITS version of DDT). Thus one must consider the scope of his task in relation to the language desired and the operational ease preferred.

Unfortunately neither LAP nor the system reported in A.I. memo No. 127 solves the problem of loading and running arbitrary binary programs jointly with LISP. Something like a runtime primitive STINK is needed for LISP, and such may have to wait for further development in the multiprogramming capabilities of the PDP-6 systems.

It is assumed that the user of PDP-6 LAP is familiar with A.I. Memoranda Nos. 116A and 144, which outline the use of the MAC PDP-6 LISP.

FORMAT OF LAP USAGE

A call to LAP is even a little more non-standard than indicated in the introduction in that not all the arguments are included in the S-expression which commences assembly -- LAP repeatedly calls READ, operating on the S-expresions read-in (from the current input device and file), until a NIL is encounterd, at which time assembly is terminated. Only after successful termination of assembly is BPORG updated and the correct flag (SUBR, FSUBR or LSUBR) inserted on the property list of the atoms which name the newly assembled functions. Thus a call to LAP would look like the sequence

```
(LAP FOO SUBR)
(DEFSYM A 1)
(HLRZ A, 0(A))
(POPJ P)
NIL
```

instead of the following written in a hypothetical style after 7090 LAP

```
(LAP ((FOO SUBR 2)
      (HLRZ A, 0(A))
      (POPJ P))
     ((HLRZ . 554_27.)
      (POPJ . 263_27.)
      (A . 1)
      (P . 14)))
```

The most serious drawback to the latter style is the strain placed on free storage, since the entire expression would have to be in core before evaluation could begin.

Hence evaluation of (LAP name indicator) or (LAP name indicator address update) begins a LAP assembly for a function with name "name" of type "indicator" (such as SUBR, FSUBR, or LSUBR) and with entry point the first location assembled into; if the second form is used, assembly begins in the core location "address" instead of BPORG. Ordinarily at assembly termination, BPORG is set to the address following the last one assembled into by LAP, but if "update" is NIL, BPORG is undisturbed.

LAP acts on the quantities it reads as follows:

QUANTITY	ACTION
NIL	Terminate assembly and return. Literal generated constants are assembled into core, symbol definitions from DEFSYM are flushed, and worthless atoms are removed from the oblist. A common error is to forget that carriage return and E-O-F are not atom break characters; NIL should be followed by a space.

atom

Assign "atom" an assembly symbol value equal to the address of the current assembly location; no additional assembly takes place. Thus one uses atoms for symbolic location tags and under certain conditions these names are entered in DDT's symbol table (see below).

(DEFSYM atom sexp . . . atom sexp)

Assign "atom " an assembly symbol value equal to (EVAL sexp); no additional asembly takes place, and these names are not entered into DDT's symbol table.

(ENTRY name2)

Sets up "name2" as a function of type indicated in the call to LAP and with enty point the current assembly location; no further assembly takes place.

(COMMENT list-fragment)

By a neat technique, no unnecessary atoms remain on the oblist after assembly; however, during assembly, there must be enough full word space to hold print names for all the atoms and to hold the numerical values of a few LISP numbers.

(EVAL sexp . . . sexp)

The expressions "sexp " to "sexp " are evaluated, in order left to right, but otherwise no assembly takes place. General computations and side effects may take place here, but caveat emptor cannot be too strongly intoned.

(SYMBOLS t-or-nil)

NIL turns off and non-NIL turns on the LAP feature which passes along symbolic location names to the job symbol table; currently all symbols so entered are treated as global, but at some time in the future this may be modified to permit flexible duplication of symbols in several programs. If the SYMBOLS pseudo-op appears anywhere in an assembly, then the names of functions thus defined will be transmitted to DDT. NOTE WELL: Although LISP atoms may be composed from upwards of 80 characters, those used as tags which are entered in the symbol table should include only upper-case letters and digits, and only the first six characters of the atom's PNAME are relevant to this feature.

(x)
(x ; list-fragment)

x (which is not among DEFSYM, ENTRY, COMMENT, EVAL, or SYMBOLS) is evaluated by LAPEVAL and the numerical result stored in the current assembly location, which is then advanced by one. For the meaning of LAPEVAL, see the next section on assembly constituents. "list-fragment" is ignored and may serve as commentary (see note above for COMMENT).

(x y)
(x y ; list-fragment)
Same as immediately above, but (LSH (LAPEVAL y) 23.) is added into the stored result.
(x y z)
(x y z ; list fragment)
Same as immediately above, but (BOOLE 1 (LAPEVAL z) 777777) is added into the stored result. Forward reference symbols may appear only in the z field; that is, if a symbol is used before it is defined, it must be used only in the address part of the instruction.
(x y z w)
(x y z w ; list-fragment)
Same as immediately above, but the numerical value of (LAPEVAL w), treated as a 36-bit quantity, is swapped left-half for right and then added into the stored result.

LAP initially checks whether or not the atom ` is a member of the list forming the assembly word, and if so sets the indirect bit (bit 13) and deletes the ` from the indicated assembly; thus an ` does not count as one of x, y, z, or w.

One notices that there is a strong similarity between LAP format and MIDAS format, an essential difference being that LAP processes assembly quantities "in order", left-to-right, to determine which is the AC field, which the address field, and which the index field. One must remember that the LISP read routine imposes a certain dissimilarity in text for the two assemblers, since "space", "comma", "left paren", and "right paren" are the only break characters for atom names. Hence spaces are necessary on both sides of a semi-colon or at-sign when they are used as described above, and the AC field may not be ommitted in instructions like (JRST 0 ADDRESS). The index field need not be enclosed in parentheses as in MIDAS, but in general there is no harm in doing so (see "anyother list" on page 7).

NORMAL AND ERROR RETURNS

Normally, after terminating assembly, LAP returns a list containing the current value of BPORG, and the names of the subroutines just assembled (there may be more than one entry for the routine assembled, the principle entry is declared in the call to LAP and others may be declared by means of the pseudo-operation ENTRY) If after assembly, some referenced symbols remain undefined, the message "UNDEF SYMBOLS", followed by the offending atoms, will be printed out. If there were any multiple-defined symbols, "AMBIG SYMBOLS" is printed along with a list of the offenders. One particular disaster caught by LAP is indicated by the message "BINARY PROGRAM SPACE EXHAUSTED".

Since LAP uses so many free variables (and for several other reasons), one should allow a call to LAP to exit by itself rather than stopping it with ^C or some other ruse.

ASSEMBLY CONSTITUENTS

Each of the parts of an assembly word (x, y, z, or w) is evaluated by LAPEVAL, in the context of the assembly. The assembly quantities whose CAR is among DEFSYM, EVAL, COMMENT, and SYMBOLS, may be termed pseudo-ops in that they do not give rise to an assembly word but merely give directions to the assembler. @ and ; are treated specially by LAP and are not considered to be assembly constituents.

QUANTITY	VALUE
number	Fixed-point numbers always evaluate to themselves. Floating-point numbers in an address field may produce Random Results.
NIL	Same as (QUOTE NIL).
*	The address of the current assembly location. Same as . in MIDAS.
atom	Except for @, ;, *, and NIL, all atoms evaluate to their assembly symbol value; i.e. (GET (QUOTE atom) (QUOTE SYM)).
(QUOTE sexp)	(MAKNUM (QUOTE sexp) (QUOTE FIXNUM)). For example, (MOVEI 1, (QUOTE (SMALL LIST))) assembles into an instruction which moves a pointer to the list (SMALL LIST) into accumulator 1.
(SPECIAL atom)	Provides a pointer to the value cell of "atom". Thus (MOVE 1, (SPECIAL FOO)) moves the value of FOO into 1 instead of a pointer to FOO, as would happen if (QUOTE FOO) were used. In addition (MOVE 1, (SPECIAL BAR)) (MOVEM 1, (SPECIAL FOO)) accomplishes in a SUBR what (SETQ FOO BAR) does in an EXPR.
(FUNCTION atom)	Essentially the same as (QUOTE atom), but is used to emphasize that "atom" is used as a function name (see section on UUO instructions).
(% x y z w)	Literal generation feature, like {x y, z (w)} in MIDAS. Assembles (x y z w) as described above and provides the address thereof. Similarly, the forms (% x), (% x y), (% x y z) may be used. A literal constant is restricted to the z-field (or address field) of a LAP instruction, but may appear nested to any finite depth. For example, (MOVE 1, (% 1.0)) moves a machine floating point number into 1, whereas (MOVEI 1, (QUOTE 1.0)) moves a LISP number.

anyotherlist

In this case LAPEVAL merely sums the LAPEVAL of each member of the list. For example, (JRST 0 (* -3)) is equivalent to JRST .-3 in MIDAS, and (MOVE A, -4(A)) assembles into the same thing as (MOVE 1 -4 A).

PDP-6 AND UUO INSTRUCTIONS

Because of the current inefficiency of the LAP symbol table, not all PDP-6 instructions are predefined, but only those listed in the appendix. Within a given call to LAP, the pseud-op DEFSYM may be used to define additional instructions as well as other local symbols which are then flushed after assembly termination. A top level function OPS is provided with LAP so that permanent definitions may be made for often used symbols. Evaluating

(OPS atom sexp . . . atom sexp)

takes the same action as DEFSYM, but these definitions are not flushed after a call to LAP (see discussion of REMLAP on page 8 under "Availability"). Permanent definitions may be temporarily reset by DEFSYM, and upon termination LAP will restore the previous values.

Four UUO (trap) instructions are predefined in LAP, which provide the means for LAP assembled code to call other LISP functions. The form for calling EXPR's, SUBR's, FEXPR's, and FSUBR's is

(CALL n (FUNCTION atom))

where n is 17 if "atom" is a FSUBR or FEXPR and the argument is in accumulator 1; n is equal to the number of arguments when calling a SUBR or EXPR and may not be greater than five. The arguments should be placed successively in accumulators 1 to 5. To call an ISUBR, place a pointer to the return address on the PDL, followed by the arguments to the function, last on top; place in accumulator 6 the negative of the number of arguments. Then do

(JCALL 16 (FUNCTION atom)).

Not only must a LAP coded routine obey these conventions, but may also expect to find its arguments, in each case, as described. Accumulators 1 through 7 are the only ones available for use within a subroutine, which is expected to return a value in accumulator 1; however accumulator 14 (P) is still used as the system push down list.

When a UUO is executed, the system will call the interpreter if "atom" does not have a SUBR or FSUBR flag on its property list; if it does, however, the system will execute a PUSHJ (in the case of CALL and CALLF) or a JRST (in the case of JCALL and JCALLF) to the appropriate code. In favorable conditions, the UUO instruction will actually be changed, in core, to a PUSHJ or JRST, but in no case is CALLF or JCALLF ever changed. The F forms are used to insure that the interpreter will be called, under conditions such as calling a FUNARG. Evaluating (NOUUO T) inhibits the instruction-modify feature so that traceing may be utilized during program checkout.

APPENDIX

SYMBOLS PRE-DEFINED IN LAP

(i) Move type instructions

MOVE MOVEI MOVEM MOVNI EXCH MOVSS

(ii) Half-word
HLRZ HRRZ HRRM HRLM HRRZS HLLZS HRLZI

(ii) Conditional branch
JRST JUMPE JUMPN JSP
SKIPN SKIPE SOJN SOJE AOSL
CAIE CAIN CAILE CAME CAMN CAMLE

(iv) Arithmetic
ADD SUB ADDB ADDI SUBI IMULI

(v) Stack
PUSH PUSHJ POP POPJ

(vi) UUO
CALL JCALL CALLF JCALLF *CALL

(vii) Miscellaneous
DPB LDB CLEARM CLEARB SETOM
TDZA TLO TRNE ANDI

(viii) Other symbol definitions
P INUM SPECBIND SPECSTR NUMVAL FLOAT FIX1A FIX2

AVAILABILITY

A symbolic version of LAP is found on the tape called LISP SYS as file ENGLSH LAP, and will sometimes reside on the COMMON disk under the same name. LAP actually bootstraps some critical portions of itself as SUBR's and requires about 250 (octal) cells of binary program space. LAP occupies a large amount of storage, and operates more quickly when there is plenty of full-word space available. A top-level function REMLAP is provided to reclaim some of the storage consumed by LAP and its peculiar atoms -- (REMLAP) will remove the EXPR's and FEXPR's (except for OPS and REMLAP) but leave permanent symbol definitions intact; (REMLAP T) REMOB's about 100 atoms peculiar to LAP and all atoms with a SYM property; (REMLAP NIL) merely removes the SYM property from any atom which has one.

A compiled version of LAP exists on the LISP SYS tape as file C LAP, which runs about twice as fast as the symbolic version (much time

is spent in READ and hence the time savings of compiled code is not so large). This version requires about 1350 (octal) cells of binary program space, and performs (REMLAP) after bootstraping in the greater part of the associated LAP functions as SUBRS. Needless to say, it is much slower in loading.

There is some possibility that LAP may be included with another group of little-used LISP routines, which will be a part of the standard LISP system and which may be excised under program control. But this is still uncertain.

EXAMPLE

```
(UREAD C LAP COM)
COM
^Q
LOADING/ LAP
BEGIN/ BOOTSTRAP
BS/ PHASE/ 2
FINISHED

(UREAD LAP XMPL DDT)
DDT
^Q
UNDEF SYMBOLS
(HLRWDSW)
AMBIG SYMBOLS
(HRLM)

^V$P

(65372 DPADD DISCNTRL)
```

See next page for listing. Although we have inadvertently defined HRLM, which is already predefined by LAP, there is nothing to be concerned about, since The message "ambig symbols" merely indicates that the symbol definition of HRLM was reset during assembly and restored to its top level state after termination. In this case no change at all was effected. The "undef symbols" message helps us to find an error -- the incorrect spelling of HLFWDSW on line 10.

The line "^V$P" is printed by DDT upon its return to LISP just after making additions to the job symbol table. The symbols transmitted were DISCNTRL, DISTOP, XIT, I, HLFWDS, and DPADD.

```
(LAP DISCNTRL SUBR)
(SYMBOLS T)
(DEFSYM HRLM 506+33 AOSE 352+33
        A 1 B 2)
        (JUMPE A, DISTOP)
        (MOVEI B, (QUOTE SUBR))
        (CALL 2 (FUNCTION GET))
        (MOVE A, -1(A))
        (HRRM A, I)
        (CLEARM 0 HLRWDSW ; 0 MEANS STORE NEXT IN LEFT)
        (MOVEI A, 30117 ; DISPLAY PARAMETER WORD)
        (JRST 0 STR)
DISTOP  (PUSHJ P, STR)
        (MOVEI A, 3000 ; STOP CHARACTER)
XIT     (JRST 0 STR)
I       (0)
HLFWDSW (0)
(SYMBOLS NIL)
(ENTRY DPADD)
        (SUBI A, 1)
        (ANDI A, 1777)
        (JUMPE B, CKX ; 0 MEANS X QUANTITY)
        (ADDI A, 220000)
        (SKIPN 0 HLFWDSW ; NONZERO MEANS Y)
        (JRST 0 SKP1)
STR     (AOSE 0 HLFWDSW)
        (JRST 0 (* 4))
        (HRRM # A,I)
        (AOSE 0 I)
        (POPJ P)
        (HRLM A, # I ; POSITION OF # IN LIST DOESNT MATTER)
        (SETOM 0 HLFWDSW)
        (POPJ P)
CKX     (ADDI A, 22000)
        (SKIPN 0 HLFWDSW)
        (JRST 0 STR)
        (AOSE 0 I)
SKP1    (AOSE 0 HLFWDSW)
        (SETOM 0 HLFWDSW)
        (JRST 0 STR)
NIL
```